CHAPTER 6

CONSTRAINT SATISFACTION PROBLEMS

In which we see how treating states as more than just little black boxes leads to new search methods and a deeper understanding of problem structure.

Chapters 3 and 4 explored the idea that problems can be solved by searching the state space: a graph where the nodes are states and the edges between them are actions. We saw that domain-specific heuristics could estimate the cost of reaching the goal from a given state, but that from the point of view of the search algorithm, each state is atomic, or indivisible—a black box with no internal structure. For each problem we need domain-specific code to describe the transitions between states.

In this chapter we break open the black box by using a **factored representation** for each state: a set of **variables**, each of which has a **value**. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or **CSP**.

Constraint satisfaction problem

CSP search algorithms take advantage of the structure of states and use *general* rather than domain-specific heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints. CSPs have the additional advantage that the actions and transition model can be deduced from the problem description.

6.1 Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, $\mathcal{X}$, $\mathcal{D}$, and $\mathcal{C}$:

$\mathcal{X}$ is a set of variables, $\{X_1, \ldots, X_n\}$.

$\mathcal{D}$ is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.

$\mathcal{C}$ is a set of constraints that specify allowable combinations of values.

A domain, D_i, consists of a set of allowable values, $\{v_1, \ldots, v_k\}$, for variable X_i. For example, a Boolean variable would have the domain $\{true, false\}$. Different variables can have different domains of different sizes. Each constraint $\mathcal{C}_j$ consists of a pair $\langle scope, rel \rangle$, where *scope* is a tuple of variables that participate in the constraint and *rel* is a **relation** that defines the values that those variables can take on. A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation. For example, if X_1 and X_2 both have the domain $\{1, 2, 3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as $\langle (X_1, X_2), \{(3,1), (3,2), (2,1)\} \rangle$ or as $\langle (X_1, X_2), X_1 > X_2 \rangle$.

Relation

CSPs deal with **assignments** of values to variables, $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that leaves some variables unassigned, and a **partial solution** is a partial assignment that is consistent. Solving a CSP is an NP-complete problem in general, although there are important subclasses of CSPs that can be solved very efficiently.

Assignments
Consistent
Complete assignment
Solution
Partial assignment
Partial solution

6.1.1 Example problem: Map coloring

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions:

$$\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}.$$

The domain of every variable is the set $D_i = \{red, green, blue\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$\begin{aligned} \mathcal{C} \;=\; & \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ & WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}. \end{aligned}$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}.$$

There are many possible solutions to this problem, such as

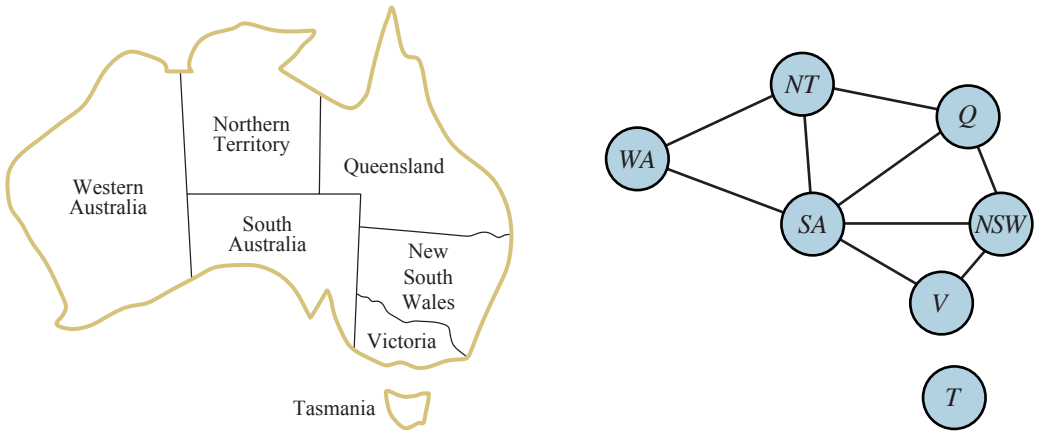
$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

It can be helpful to visualize a CSP as a **constraint graph**, as shown in Figure 6.1(b). The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.

Constraint graph

Why formulate a problem as a CSP? One reason is that the CSPs yield a natural representation for a wide variety of problems; it is often easy to formulate a problem as a CSP. Another is that years of development work have gone into making CSP solvers fast and efficient. A third is that a CSP solver can quickly prune large swathes of the search space that an atomic state-space searcher cannot. For example, once we have chosen $\{SA = blue\}$ in the Australia problem, we can conclude that none of the five neighboring variables can take on the value *blue*. A search procedure that does not use constraints would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraints we have only $2^5 = 32$ assignments to consider, a reduction of 87%.

In atomic state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment violates a constraint, we can immediately discard further refinements of the partial assignment. Furthermore, we can see *why* the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for atomic state-space search can be solved quickly when formulated as a CSP.



(a) (b)

Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

6.1.2 Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

$$\begin{aligned} \mathcal{X} = \{&Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, \\ &Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}. \end{aligned}$$

Precedence constraint

Next, we represent **precedence constraints** between individual tasks. Whenever a task T_1 must occur before task T_2, and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of the form

$$T_1 + d_1 \leq T_2 .$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$\begin{aligned} &Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF}; \\ &Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}. \end{aligned}$$

Next we say that for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF};$$
$$Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF};$$
$$Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB};$$
$$Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB}.$$

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:

Disjunctive constraint

$$(Axle_F + 10 \leq Axle_B) \quad \textbf{or} \quad (Axle_B + 10 \leq Axle_F).$$

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that $Axle_F$ and $Axle_B$ can take on.

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except *Inspect* we add a constraint of the form $X + d_X \leq Inspect$. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i = \{0, 1, 2, 3, \ldots, 30\}.$$

This particular problem is trivial to solve, but CSPs have been successfully applied to job-shop scheduling problems like this with thousands of variables.

6.1.3 Variations on the CSP formalism

The simplest kind of CSP involves variables that have **discrete**, **finite domains**. Map-coloring problems and scheduling with time limits are both of this kind. The 8-queens problem (Figure 4.3) can also be viewed as a finite-domain CSP, where the variables $Q_1, \ldots, Q_8$ correspond to the queens in columns 1 to 8, and the domain of each variable specifies the possible row numbers for the queen in that column, $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$. The constraints say that no two queens can be in the same row or diagonal.

Discrete domain
Finite domain

A discrete domain can be **infinite**, such as the set of integers or strings. (If we didn't put a deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.) With infinite domains, we must use implicit constraints like $T_1 + d_1 \leq T_2$ rather than explicit tuples of values. Special solution algorithms (which we do not discuss here) exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables—the problem is undecidable.

Infinite

Linear constraints

Nonlinear constraints

Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on. These problems constitute an important area of applied mathematics.

Continuous domains

Unary constraint

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; we can express that with the unary constraint $\langle (SA), SA \neq green \rangle$. (The initial specification of the domain of a variable can also be seen as a unary constraint.)

Binary constraint

Binary CSP

A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint. A **binary CSP** is one with only unary and binary constraints; it can be represented as a constraint graph, as in Figure 6.1(b).

We can also define higher-order constraints. The ternary constraint $Between(X,Y,Z)$, for example, can be defined as $\langle (X,Y,Z), X < Y < Z \textbf{ or } X > Y > Z \rangle$.

Global constraint

A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because a global constraint need not involve *all* the variables in a problem). One of the most common global constraints is *Alldiff*, which says that all of the variables involved in the constraint must have different values. In Sudoku problems (see Section 6.2.6), all variables in a row, column, or 3×3 box must satisfy an *Alldiff* constraint.

Cryptarithmetic

Another example is provided by **cryptarithmetic** puzzles (Figure 6.2(a)). Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint $Alldiff(F,T,U,W,R,O)$. The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

$$\begin{aligned} O + O &= R + 10 \cdot C_1 \\ C_1 + W + W &= U + 10 \cdot C_2 \\ C_2 + T + T &= O + 10 \cdot C_3 \\ C_3 &= F\,, \end{aligned}$$

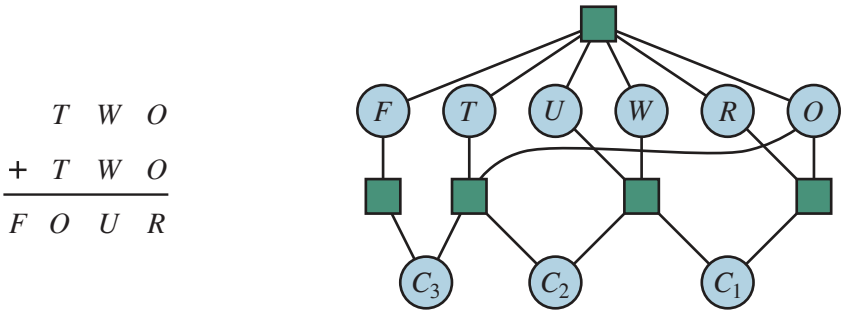
where C_1, C_2, and C_3 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints—constraints involving n variables.

Constraint hypergraph

Alternatively, as Exercise 6.NARY asks you to prove, every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced. This means that we could transform any CSP into one with only binary constraints—which certainly makes the life of the algorithm designer simpler. Another way to convert an n-ary CSP to a binary one is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and one binary constraint for each pair of constraints in the original graph that share variables.

Dual graph

For example, consider a CSP with the variables $\mathcal{X} = \{X,Y,Z\}$, each with the domain $\{1,2,3,4,5\}$, and with the two constraints $C_1 : \langle (X,Y,Z), X+Y=Z \rangle$ and $C_2 : \langle (X,Y), X+1=Y \rangle$. Then the dual graph would have the variables $\mathcal{X} = \{C_1, C_2\}$, where the domain of the C_1 variable in the dual graph is the set of $\{(x_i, y_j, z_k)\}$ tuples from the C_1 constraint in the original problem, and similarly the domain of C_2 is the set of $\{(x_i, y_j)\}$ tuples. The dual graph has the binary constraint $\langle (C_1, C_2), R_1 \rangle$, where R_1 is a new relation that defines the constraint between C_1 and C_2; in this case it would be $R_1 = \{((1,2,3),(1,2)),((2,3,5),(2,3))\}$.



(a) (b)

Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns from right to left.

There are however two reasons why we might prefer a global constraint such as *Alldiff* rather than a set of binary constraints. First, it is easier and less error-prone to write the problem description using *Alldiff*. Second, it is possible to design special-purpose inference algorithms for global constraints that are more efficient than operating with primitive constraints. We describe these inference algorithms in Section 6.2.5.

Preference constraints

The constraints we have described so far have all been absolute constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference constraints** indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one.

Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constrained optimization problem**, or COP. Linear programs are one class of COPs.

Constrained optimization problem

6.2 Constraint Propagation: Inference in CSPs

An atomic state-space search algorithm makes progress in only one way: by expanding a node to visit the successors. A CSP algorithm has choices. It can generate successors by choosing a new variable assignment, or it can do a specific type of inference called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which

Constraint propagation

in turn can reduce the legal values for another variable, and so on. The idea is that this will leave fewer choices to consider when we make the next choice of a variable assignment. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Local consistency

The key idea is **local consistency**. If we treat each variable as a node in a graph (see Figure 6.1(b)) and each binary constraint as an edge, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

6.2.1 Node consistency

Node consistency

A single variable (corresponding to a node in the CSP graph) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem (Figure 6.1) where South Australians dislike green, the variable SA starts with domain $\{red, green, blue\}$, and we can make it node consistent by eliminating $green$, leaving SA with the reduced domain $\{red, blue\}$. We say that a graph is node-consistent if every variable in the graph is node-consistent.

It is easy to eliminate all the unary constraints in a CSP by reducing the domain of variables with unary constraints at the start of the solving process. As mentioned earlier, it is also possible to transform all n-ary constraints into binary ones. Because of this, some CSP solvers work with only binary constraints, expecting the user to eliminate the other constraints ahead of time. We make that assumption for the rest of this chapter, except where noted.

6.2.2 Arc consistency

Arc consistency

A variable in a CSP is **arc-consistent**[1] if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A graph is arc-consistent if every variable is arc-consistent with every other variable. For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of decimal digits. We can write this constraint explicitly as

$$\langle (X, Y), \{(0,0), (1,1), (2,4), (3,9)\} \rangle .$$

To make X arc-consistent with respect to Y, we reduce X's domain to $\{0, 1, 2, 3\}$. If we also make Y arc-consistent with respect to X, then Y's domain becomes $\{0, 1, 4, 9\}$, and the whole CSP is arc-consistent. On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (SA, WA):

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\} .$$

No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

The most popular algorithm for enforcing arc consistency is called AC-3 (see Figure 6.3). To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. Initially, the queue contains all the arcs in the CSP. (Each binary constraint becomes two arcs, one in each direction.) AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue

[1] We have been using the term "edge" rather than "arc," so it would make more sense to call this "edge-consistent," but the name "arc-consistent" is historical.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it was the third version developed in the paper.

and makes X_i arc-consistent with respect to X_j. If this leaves D_i unchanged, the algorithm just moves on to the next arc. But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i. We need to do that because the change in D_i might enable further reductions in D_k, even if we have previously considered X_k. If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will be faster to search because its variables have smaller domains. In some cases, it solves the problem completely (by reducing every domain to size 1) and in others it proves that no solution exists (by reducing some domain to size 0).

The complexity of AC-3 can be analyzed as follows. Assume a CSP with n variables, each with domain size at most d, and with c binary constraints (arcs). Each arc (X_k, X_i) can be inserted in the queue only d times because X_i has at most d values to delete. Checking consistency of an arc can be done in $O(d^2)$ time, so we get $O(cd^3)$ total worst-case time.

6.2.3 Path consistency

Suppose we are to color the map of Australia with just two colors, red and blue. Arc consistency does nothing because every constraint can be satisfied individually with red at one end and blue at the other. But clearly there is no solution to the problem: because Western Australia, Northern Territory, and South Australia all touch each other, we need at least three colors for them alone.

Path consistency

Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency. **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints (if any) on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. The name refers to the overall consistency of the path from X_i to X_j with X_m in the middle.

Let's see how path consistency fares in coloring the Australia map with two colors. We will make the set $\{WA, SA\}$ path-consistent with respect to NT. We start by enumerating the consistent assignments to the set. In this case, there are only two: $\{WA = red, SA = blue\}$ and $\{WA = blue, SA = red\}$. We can see that with both of these assignments NT can be neither *red* nor *blue* (because it would conflict with either WA or SA). Because there is no valid choice for NT, we eliminate both assignments, and we end up with no valid assignments for $\{WA, SA\}$. Therefore, we know that there can be no solution to this problem.

6.2.4 K-consistency

K-consistency

Stronger forms of propagation can be defined with the notion of **k-consistency**. A CSP is k-consistent if, for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable. 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency. 2-consistency is the same as arc consistency. For binary constraint graphs, 3-consistency is the same as path consistency.

Strongly k-consistent

A CSP is **strongly k-consistent** if it is k-consistent and is also $(k-1)$-consistent, $(k-2)$-consistent, ... all the way down to 1-consistent. Now suppose we have a CSP with n nodes and make it strongly n-consistent (i.e., strongly k-consistent for $k=n$). We can then solve the problem as follows: First, we choose a consistent value for X_1. We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on. For each variable X_i, we need only search through the d values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$. The total run time is only $O(n^2 d)$.

Of course, there is no free lunch: constraint satisfaction is NP-complete in general, and any algorithm for establishing n-consistency must take time exponential in n in the worst case. Worse, n-consistency also requires space that is exponential in n. In practice, determining the appropriate level of consistency checking is mostly an empirical science. Computing 2-consistency is common, and 3-consistency less common.

6.2.5 Global constraints

Remember that a **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the *Alldiff* constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem above and Sudoku puzzles below). One simple form of inconsistency detection for *Alldiff* constraints works as follows: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

This method can detect the inconsistency in the assignment $\{WA = red, NSW = red\}$ for Figure 6.1. Notice that the variables SA, NT, and Q are effectively connected by an *Alldiff* constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the domains of SA, NT, and Q are all reduced to $\{green, blue\}$. That is, we have three variables and only two colors, so the *Alldiff* constraint is violated. Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints.

Resource constraint

Another important higher-order constraint is the **resource constraint**, sometimes called the *Atmost* constraint. For example, in a scheduling problem, let $P_1, \ldots, P_4$ denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as $Atmost(10, P_1, P_2, P_3, P_4)$. We can detect an inconsistency simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain $\{3,4,5,6\}$, the *Atmost* constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain $\{2,3,4,5,6\}$, the values 5 and 6 can be deleted from each domain.

Bounds propagation

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods. Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. For example, in an airline-scheduling problem, let's suppose there are two flights, F_1 and F_2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on flights F_1 and F_2 are then

$$D_1 = [0, 165] \quad \text{and} \quad D_2 = [0, 385] .$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$. Propagating bounds constraints, we reduce the domains to

$$D_1 = [35, 165] \quad \text{and} \quad D_2 = [255, 385] .$$

Bounds-consistent

We say that a CSP is **bounds-consistent** if for every variable X, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y. This kind of bounds propagation is widely used in practical constraint problems.

6.2.6 Sudoku

Sudoku

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not realize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box (see Figure 6.4). A row, column, or box is called a **unit**.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

The Sudoku puzzles that appear in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, a CSP solver can handle thousands of puzzles per second.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names *A1* through *A9* for the top row (left to right), down to *I1* through *I9* for the bottom row. The empty squares have the domain $\{1,2,3,4,5,6,7,8,9\}$ and the pre-filled squares have a domain consisting of a single value. In addition, there are 27 different *Alldiff* constraints, one for each unit (row, column, and box of 9 squares):

$$\begin{aligned}
&\mathit{Alldiff}(A1,A2,A3,A4,A5,A6,A7,A8,A9)\\
&\mathit{Alldiff}(B1,B2,B3,B4,B5,B6,B7,B8,B9)\\
&\ldots\\
&\mathit{Alldiff}(A1,B1,C1,D1,E1,F1,G1,H1,I1)\\
&\mathit{Alldiff}(A2,B2,C2,D2,E2,F2,G2,H2,I2)\\
&\ldots\\
&\mathit{Alldiff}(A1,A2,A3,B1,B2,B3,C1,C2,C3)\\
&\mathit{Alldiff}(A4,A5,A6,B4,B5,B6,C4,C5,C6)\\
&\ldots
\end{aligned}$$

Let us see how far arc consistency can take us. Assume that the *Alldiff* constraints have been expanded into binary constraints (such as $A1 \neq A2$) so that we can apply the AC-3 algorithm directly. Consider variable *E6* from Figure 6.4(a)—the empty square between the 2 and the 8 in the middle box. From the constraints in the box, we can remove 1, 2, 7, and 8 from *E6*'s domain. From the constraints in its column, we can eliminate 5, 6, 2, 8, 9, and 3 (although 2 and 8 were already removed). That leaves *E6* with a domain of $\{4\}$; in other words, we know the answer for *E6*. Now consider variable *I6*—the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate 5, 6, 2, 4 (since we now know *E6* must be 4), 8, 9, and 3. We eliminate 1 by arc consistency with *I5*,

and we are left with only the value 7 in the domain of *I6*. Now there are 8 known values in column 6, so arc consistency can infer that *A6* must be 1. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value, as shown in Figure 6.4(b).

Of course, Sudoku would soon lose its appeal if every puzzle could be solved by a mechanical application of AC-3, and indeed AC-3 works only for the easiest Sudoku puzzles. Slightly harder ones can be solved by PC-2, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle. To solve the hardest puzzles and to make efficient progress, we will have to be more clever.

Indeed, the appeal of Sudoku puzzles for the human solver is the need to be resourceful in applying more complex inference strategies. Aficionados give them colorful names, such as "naked triples." That strategy works as follows: in any unit (row, column or box), find three squares that each have a domain that contains the same three numbers or a subset of those numbers. For example, the three domains might be $\{1,8\}$, $\{3,8\}$, and $\{1,3,8\}$. From that we don't know which square contains 1, 3, or 8, but we do know that the three numbers must be distributed among the three squares. Therefore we can remove 1, 3, and 8 from the domains of every *other* square in the unit.

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 *Alldiff* constraints. But beyond that, all the strategies—arc consistency, path consistency, and so on—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of *Alldiff* constraints and is not specific to Sudoku *per se*. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

6.3 Backtracking Search for CSPs

Sometimes we can finish the constraint propagation process and still have variables with multiple possible values. In that case we have to **search** for a solution. In this section we cover backtracking search algorithms that work on partial assignments; in the next section we look at local search algorithms over complete assignments.

Consider how a standard depth-limited search (from Chapter 3) could solve CSPs. A state would be a partial assignment, and an action would extend the assignment, adding, say, $NSW = red$ or $SA = blue$ for the Australia map-coloring problem. For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n. But notice that the branching factor at the top level would be nd because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on for n levels. So the tree has $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

We can get back that factor of $n!$ by recognizing a crucial property of CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions does not matter. In CSPs, it makes no difference if we first assign $NSW = red$ and then $SA = blue$, or the other way around. Therefore, we need only consider a *single* variable at each node in the search tree. At the root we might make a choice between $SA = red$, $SA = green$, and

Commutativity

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, implement the general-purpose heuristics discussed in Section 6.3.1. The INFERENCE function can optionally impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are retracted and a new value is tried.

$SA = blue$, but we would never choose between $NSW = red$ and $SA = blue$. With this restriction, the number of leaves is d^n, as we would hope. At each level of the tree we do have to choose which variable we will deal with, but we never have to backtrack over that choice.

Figure 6.5 shows a backtracking search procedure for CSPs. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call. If the call succeeds, the solution is returned, and if it fails, the assignment is restored to the previous state, and we try the next value. If no value works then we return failure. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order $WA, NT, Q, \ldots$.

Notice that BACKTRACKING-SEARCH keeps only a single representation of a state (assignment) and alters that representation rather than creating new ones (see page 80).

Whereas the uninformed search algorithms of Chapter 3 could be improved only by supplying them with *domain-specific* heuristics, it turns out that backtracking search can be improved using *domain-independent* heuristics that take advantage of the factored representation of CSPs. In the following four sections we show how this is done:

- (6.3.1) Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
- (6.3.2) What inferences should be performed at each step in the search (INFERENCE)?
- (6.3.3) Can we BACKTRACK more than one step when appropriate?
- (6.3.4) Can we save and reuse partial results from the search?

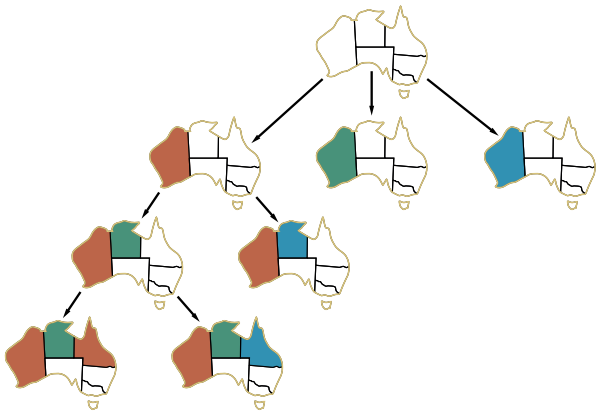


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

6.3.1 Variable and value ordering

The backtracking algorithm contains the line

$$var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp, assignment).$$

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is static ordering: choose the variables in order, $\{X_1, X_2, \ldots\}$. The next simplest is to choose randomly. Neither strategy is optimal. For example, after the assignments for $WA = red$ and $NT = green$ in Figure 6.6, there is only one possible value for SA, so it makes sense to assign $SA = blue$ next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced.

This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum-remaining-values** (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by orders of magnitude, although the results vary depending on the problem.

Minimum-remaining-values

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. In Figure 6.1, SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has degree 0. In fact, once SA is chosen, applying the degree heuristic solves the problem without any false steps—you can choose *any* consistent color at each choice point and still arrive at a solution with no backtracking. The minimum-remaining-values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.

Degree heuristic

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. The **least-constraining-value** heuristic is effective for this. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint

Least-constraining-value

graph. For example, suppose that in Figure 6.1 we have generated the partial assignment with $WA = red$ and $NT = green$ and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

Why should variable selection be fail-first, but value selection be fail-last? Every variable has to be assigned eventually, so by choosing the ones that are likely to fail first, we will on average have fewer successful assignments to backtrack over. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

6.3.2 Interleaving search and inference

We saw how AC-3 can reduce the domains of variables *before* we begin the search. But inference can be even more powerful *during* the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

Forward checking

One of the simplest forms of inference is called **forward checking**. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after $WA = red$ and $Q = green$ are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. A second point to notice is that after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm backtracks immediately.

For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider Figure 6.7 after assigning $\{WA = red\}$. Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those variables next, and then all the other variables will fall into place. That's exactly what happens with MRV: NT and SA each have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it doesn't look ahead far enough. For example, consider the $Q = green$ row of Figure 6.7. We've made WA and Q arc-consistent, but we've left both NT and SA with blue as their only possible value, which is an inconsistency, since they are neighbors.

Maintaining Arc Consistency

The algorithm called MAC (for **Maintaining Arc Consistency**) detects inconsistencies like this. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3

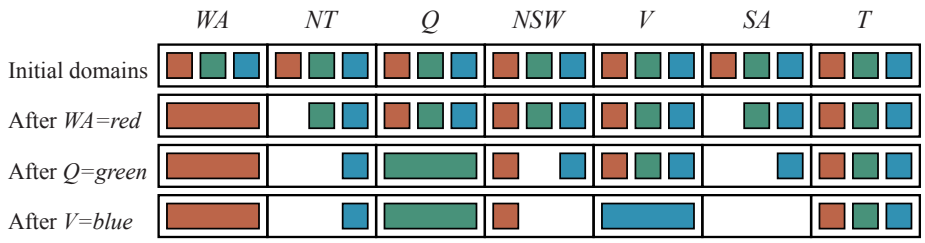


Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$ is assigned, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$ is assigned, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

6.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm in Figure 6.5 has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited. In this subsection, we consider better possibilities.

Chronological backtracking

Consider what happens when we apply simple backtracking in Figure 6.1 with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generated the partial assignment $\{Q = red, NSW = green, V = blue, T = red\}$. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly help in resolving the problem with South Australia.

A more intelligent approach is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of SA impossible. To do this, we will keep track of a set of assignments that are in conflict with some value for SA. The set (in this case $\{Q = red, NSW = green, V = blue\}$), is called the **conflict set** for SA. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

Conflict set

Backjumping

The sharp-eyed reader may have noticed that forward checking can supply the conflict set with no extra work: whenever forward checking based on an assignment $X = x$ deletes a value from Y's domain, it should add $X = x$ to Y's conflict set. If the last value is deleted from Y's domain, then the assignments in the conflict set of Y are added to the conflict set of X. That is, we now know that $X = x$ leads to a contradiction (in Y), and thus a different assignment should be tried for X.

The eagle-eyed reader may have noticed something odd: backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that *every* branch pruned by backjumping is also pruned by forward checking. Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking, such as MAC—you need only do one or the other.

Despite the observations of the preceding paragraph, the idea behind backjumping remains a good one: to backtrack based on the reasons for failure. Backjumping notices failure when a variable's domain becomes empty, but in many cases a branch is doomed long before this occurs. Consider again the partial assignment $\{WA = red, NSW = red\}$ (which, from our earlier discussion, is inconsistent). Suppose we try $T = red$ next and then assign NT, Q, V, SA. We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now, the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables—NT doesn't have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together*, failed because of a set of preceding variables, which must be those variables that directly conflict with the four.

This leads to a different–and deeper–notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables*, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

Conflict-directed backjumping

We must now explain how these new conflict sets are computed. The method is in fact quite simple. The "terminal" failure of a branch of the search always occurs because a variable's domain becomes empty; that variable has a standard conflict set. In our example, SA fails, and its conflict set is (say) $\{WA, NT, Q\}$. We backjump to Q, and Q *absorbs* the conflict set from SA (minus Q itself, of course) into its own direct conflict set, which is $\{NT, NSW\}$; the new conflict set is $\{WA, NT, NSW\}$. That is, there is no solution from Q onward, given the preceding assignment to $\{WA, NT, NSW\}$. Therefore, we backtrack to NT, the most recent of these. NT absorbs $\{WA, NT, NSW\} - \{NT\}$ into its own direct conflict set $\{WA\}$, giving $\{WA, NSW\}$ (as stated in the previous paragraph). Now the algorithm backjumps to NSW, as we would hope. To summarize: let X_j be the current variable, and let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $conf(X_j)$ and recompute the conflict set for X_i as follows:

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}.$$

6.3.4 Constraint learning

When we reach a contradiction, backjumping can tell us how far to back up, so we don't waste time changing variables that won't fix the problem. But we would also like to avoid running into the same problem again. When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem. **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP to forbid this combination of assignments or by keeping a separate cache of no-goods.

Constraint learning

No-good

For example, consider the state $\{WA = red, NT = green, Q = blue\}$ in the bottom row of Figure 6.6. Forward checking can tell us this state is a no-good because there is no valid assignment to SA. In this particular case, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again. But suppose that the search tree in Figure 6.6 were actually part of a larger search tree that started by first assigning values for V and T. Then it would be worthwhile to record $\{WA = red, NT = green, Q = blue\}$ as a no-good because we are going to run into the same problem again for each possible set of assignments to V and T.

No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

6.4 Local Search for CSPs

Local search algorithms (see Section 4.1) turn out to be very effective in solving many CSPs. They use a complete-state formulation (as introduced in Section 4.1.1) where each state assigns a value to every variable, and the search changes the value of one variable at a time. As an example, we'll use the 8-queens problem, as defined as a CSP on page 183. In Figure 6.8 we start on the left with a complete assignment to the 8 variables; typically this will violate several constraints. We then randomly choose a conflicted variable, which turns out to be Q_8, the rightmost column. We'd like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic.

Min-conflicts

In the figure we see there are two rows that only violate one constraint; we pick $Q_8 = 3$ (that is, we move the queen to the 8th column, 3rd row). On the next iteration, in the middle board of the figure, we select Q_6 as the variable to change, and note that moving the queen to the 8th row results in no conflicts. At this point there are no more conflicted variables, so we have a solution. The algorithm is shown in Figure 6.9.[2]

Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly *independent of problem size*. It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment). This remarkable observation was the stimulus leading to a great deal of research in the 1990s on local search and the distinction between easy and hard problems, which we take up in Section 7.6.3. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective. The landscape of a CSP under the min-conflicts heuristic usually has a series of plateaus. There may be millions of variable assignments that are only one conflict away from a solution. Plateau search—allowing sideways moves to another state with the same score—can help local search find its way off this

[2] Local search can easily be extended to constrained optimization problems (COPs). In that case, all the techniques for hill climbing and simulated annealing can be applied to optimize the objective function.

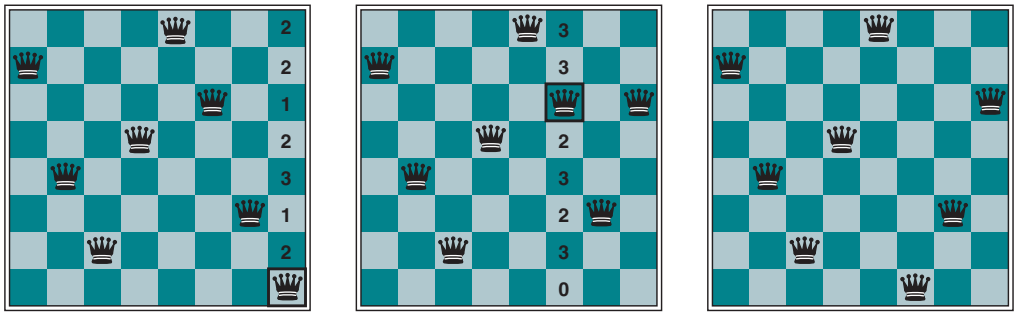


Figure 6.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or *failure*
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for *csp*
  for *i* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value *v* for *var* that minimizes CONFLICTS(*csp*, *var*, *v*, *current*)
    set *var* = *value* in *current*
  return *failure*

Figure 6.9 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

plateau. This wandering on the plateau can be directed with a technique called **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those states. Simulated annealing can also be used to escape from plateaus.

Constraint weighting

Another technique called **constraint weighting** aims to concentrate the search on the important constraints. Each constraint is given a numeric weight, initially all 1. At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment. This has two benefits: it adds topography to plateaus, making sure that it is possible to improve from the current state, and it also adds learning: over time the difficult constraints are assigned higher weights.

Another advantage of local search is that it can be used in an online setting (see Section 4.5) when the problem changes. Consider a scheduling problem for an airline's weekly flights. The schedule may involve thousands of flights and tens of thousands of personnel

assignments, but bad weather at one airport can render the schedule infeasible. We would like to repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule. A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

6.5 The Structure of Problems

In this section, we examine ways in which the *structure* of the problem, as represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here also apply to other problems besides CSPs, such as probabilistic reasoning.

The only way we can possibly hope to deal with the vast real world is to decompose it into subproblems. Looking again at the constraint graph for Australia (Figure 6.1(b), repeated as Figure 6.12(a)), one fact stands out: Tasmania is not connected to the mainland.[3] Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map.

Independent subproblems

Independence can be ascertained simply by finding **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Why is this important? Suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve, where d is the size of the domain. Hence, the total work is $O(d^c n/c)$, which is *linear* in n; without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with 100 variables into four subproblems reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Connected component

Completely independent subproblems are delicious, then, but rare. Fortunately, some other graph structures are also easy to solve. For example, a constraint graph is a **tree** when any two variables are connected by only one path. We will show that *any tree-structured CSP can be solved in time linear in the number of variables.*[4] The key is a new notion of consistency, called **directional arc consistency** or DAC. A CSP is defined to be directional arc-consistent under an ordering of variables $X_1, X_2, \ldots, X_n$ if and only if every X_i is arc-consistent with each X_j for $j > i$.

Directional arc consistency

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a **topological sort**. Figure 6.10(a) shows a sample tree and (b) shows one possible ordering. Any tree with n nodes has $n-1$ edges, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$. Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each edge from a parent to its child is arc-consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child. That means we won't

Topological sort

[3] A careful cartographer or patriotic Tasmanian might object that Tasmania should not be colored the same as its nearest mainland neighbor, to avoid the impression that it *might* be part of that state.

[4] Sadly, very few regions of the world have tree-structured maps, although Sulawesi comes close.

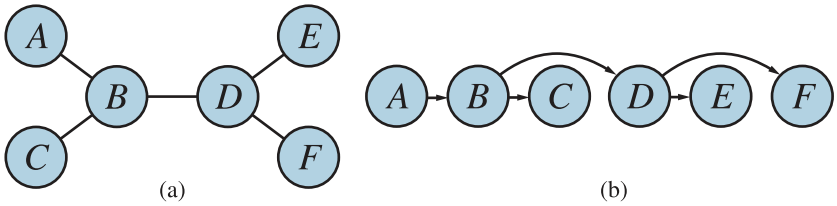


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or *failure*
  inputs: *csp*, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  assignment $\leftarrow$ an empty assignment
  root $\leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, *root*)
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for $i = 1$ **to** n **do**
    assignment$[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

have to backtrack; we can move linearly through the variables. The complete algorithm is shown in Figure 6.11.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be *reduced* to trees somehow. There are two ways to do this: by removing nodes (Section 6.5.1) or by collapsing nodes together (Section 6.5.2).

6.5.1 Cutset conditioning

The first way to reduce a constraint graph to a tree involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure 6.12(a). Without South Australia, the graph would become a tree, as in (b). Fortunately, we can delete South Australia (in the graph, not the country) by fixing a value for *SA* and deleting from the domains of the other variables any values that are inconsistent with the value chosen for *SA*.

Now, any solution for the CSP after *SA* and its constraints are removed will be consistent with the value chosen for *SA*. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm

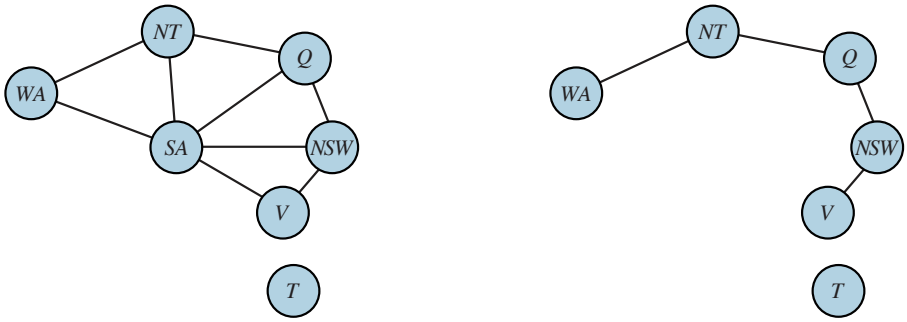


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) After the removal of *SA*, the constraint graph becomes a forest of two trees.

given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring), the value chosen for *SA* could be the wrong one, so we would need to try each possible value. The general algorithm is as follows:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**. Cycle cutset
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) if the remaining CSP has a solution, return it together with the assignment for S.

If the cycle cutset has size c, then the total run time is $O(d^c \cdot (n-c)d^2)$: we have to try each of the d^c combinations of values for the variables in S, and for each combination we must solve a tree problem of size $n-c$. If the graph is "nearly a tree," then c will be small and the savings over straight backtracking will be huge—for our 100-Boolean-variable example, if we could find a cutset of size $c=20$, this would get us down from the lifetime of the Universe to a few minutes. In the worst case, however, c can be as large as $(n-2)$. Finding the *smallest* cycle cutset is NP-hard, but several efficient approximation algorithms are known. The overall algorithmic approach is called **cutset conditioning**; it comes up again in Chapter 13, where it is used for reasoning about probabilities. Cutset conditioning

6.5.2 Tree decomposition

The second way to reduce a constraint graph to a tree is based on constructing a **tree decomposition** of the constraint graph: a transformation of the original graph into a tree where each node in the tree consists of a set of variables, as in Figure 6.13. A tree decomposition must satisfy these three requirements: Tree decomposition

- Every variable in the original problem appears in at least one of the tree nodes.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
- If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes.

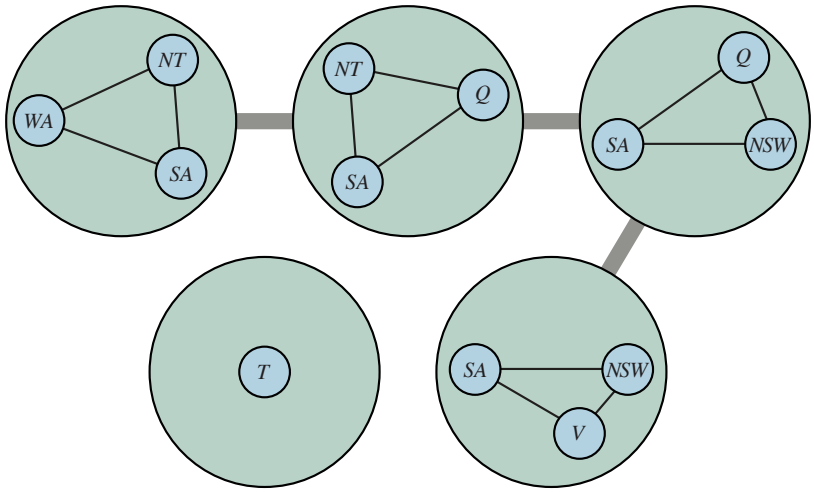


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

The first two conditions ensure that all the variables and constraints are represented in the tree decomposition. The third condition seems rather technical, but allows us to say that any variable from the original problem must have the same value wherever it appears: the constraints in the tree say that a variable in one node of the tree must have the same value as the corresponding variable in the adjacent node in the tree. For example, *SA* appears in all four of the connected nodes in Figure 6.13, so each edge in the tree decomposition therefore includes the constraint that the value of *SA* in one node must be the same as the value of *SA* in the next. You can verify from Figure 6.12 that this decomposition makes sense.

Once we have a tree-structured graph, we can apply TREE-CSP-SOLVER to get a solution in $O(nd^2)$ time, where n is the number of tree nodes and d is the size of the largest domain. But note that in the tree, a domain is a set of *tuples* of values, not just individual values.

For example, the top left node in Figure 6.13 represents, at the level of the original problem, a subproblem with variables $\{WA, NT, SA\}$, domain $\{red, green, blue\}$, and constraints $WA \neq NT, SA \neq NT, WA \neq SA$. At the level of the tree, the node represents a single variable, which we can call *SANTWA*, whose value must be a three-tuple of colors, such as $(red, green, blue)$, but not $(red, red, blue)$, because that would violate the constraint $SA \neq NT$ from the original problem. We can then move from that node to the adjacent one, with the variable we can call *SANTQ*, and find that there is only one tuple, $(red, green, blue)$, that is consistent with the choice for *SANTWA*. The exact same process is repeated for the next two nodes, and independently we can make any choice for T.

We can solve any tree decomposition problem in $O(nd^2)$ time with TREE-CSP-SOLVER, which will be efficient as long as d remains small. Going back to our example with 100 Boolean variables, if each node has 10 variables, then $d = 2^{10}$ and we should be able to solve the problem in seconds. But if there is a node with 30 variables, it would take centuries.

A given graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. (Putting all the variables into one node is technically a tree, but is not helpful.) The **tree width** of a tree decomposition of a graph is

Tree width

one less than the size of the largest node; the tree width of the graph itself is defined to be the minimum width among all its tree decompositions. If a graph has tree width w then the problem can be solved in $O(nd^{w+1})$ time given the corresponding tree decomposition. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.*

Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice. Which is better: the cutset decomposition with time $O(d^c \cdot (n-c)d^2)$, or the tree decomposition with time $O(nd^{w+1})$? Whenever you have a cycle-cutset of size c, there is also a tree width of size $w < c+1$, and it may be far smaller in some cases. So time consideration favors tree decomposition, but the advantage of the cycle-cutset approach is that it can be executed in linear memory, while tree decomposition requires memory exponential in w.

6.5.3 Value symmetry

So far, we have looked at the structure of the constraint graph. There can also be important structure in the *values* of variables, or in the structure of the constraint relations themselves. Consider the map-coloring problem with d colors. For every consistent solution, there is actually a set of $d!$ solutions formed by permuting the color names. For example, on the Australia map we know that WA, NT, and SA must all have different colors, but there are $3! = 6$ ways to assign three colors to three regions. This is called **value symmetry**. We would like to reduce the search space by a factor of $d!$ by breaking the symmetry in assignments. We do this by introducing a **symmetry-breaking constraint**. For our example, we might impose an arbitrary ordering constraint, $NT < SA < WA$, that requires the three values to be in alphabetical order. This constraint ensures that only one of the $d!$ solutions is possible: $\{NT = blue, SA = green, WA = red\}$.

Value symmetry

Symmetry-breaking constraint

For map coloring, it was easy to find a constraint that eliminates the symmetry. In general it is NP-hard to eliminate all symmetry, but breaking value symmetry has proved to be important and effective on a wide range of problems.

Summary

- **Constraint satisfaction problems** (CSPs) represent a state with a set of variable/value pairs and represent the conditions for a solution by a set of constraints on the variables. Many important real-world problems can be described as CSPs.
- A number of **inference** techniques use the constraints to rule out certain variable assignments. These include node, arc, path, and k-consistency.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- The **minimum-remaining-values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in deciding which value to try first for a given variable. Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem. **Constraint learning** records the conflicts as they are encountered during search in order to avoid the same conflict later in the search.

- Local search using the **min-conflicts** heuristic has also been applied to constraint satisfaction problems with great success.
- The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is quite efficient (requiring only linear memory) if a small cutset can be found. **Tree decomposition** techniques transform the CSP into a tree of subproblems and are efficient if the **tree width** of the constraint graph is small; however they need memory exponential in the tree width of the constraint graph. Combining cutset conditioning with tree decomposition can allow a better tradeoff of memory versus time.

Bibliographical and Historical Notes

Diophantine equations

The Greek mathematician Diophantus (c. 200–284) presented and solved problems involving algebraic constraints on equations, although he didn't develop a generalized methodology. We now call equations over integer domains **Diophantine equations**. The Indian mathematician Brahmagupta (c. 650) was the first to show a general solution over the domain of integers for the equation $ax + by = c$. Systematic methods for solving linear equations by variable elimination were studied by Gauss (1829); the solution of linear inequality constraints goes back to Fourier (1827).

Finite-domain constraint satisfaction problems also have a long history. For example, **graph coloring** (of which map coloring is a special case) is an old problem in mathematics. The four-color conjecture (that every planar graph can be colored with four or fewer colors) was first made by Francis Guthrie, a student of De Morgan, in 1852. It resisted solution—despite several published claims to the contrary—until a proof was devised by Appel and Haken (1977) (see the book *Four Colors Suffice* (Wilson, 2004)). Purists were disappointed that part of the proof relied on a computer, so Georges Gonthier (2008), using the COQ theorem prover, derived a formal proof that Appel and Haken's proof program was correct.

Specific classes of constraint satisfaction problems occur throughout the history of computer science. One of the most influential early examples was SKETCHPAD (Sutherland, 1963), which solved geometric constraints in diagrams and was the forerunner of modern drawing programs and CAD tools. The identification of CSPs as a *general* class is due to Ugo Montanari (1974). The reduction of higher-order CSPs to purely binary CSPs with auxiliary variables (see Exercise 6.NARY) is due originally to the 19th-century logician Charles Sanders Peirce. It was introduced into the CSP literature by Dechter (1990b) and was elaborated by Bacchus and van Beek (1998). CSPs with preferences among solutions are studied widely in the optimization literature; see Bistarelli *et al.* (1997) for a generalization of the CSP framework to allow for preferences.

Constraint propagation methods were popularized by Waltz's (1975) success on polyhedral line-labeling problems for computer vision. Waltz showed that in many problems, propagation completely eliminates the need for backtracking. Montanari (1974) introduced the notion of constraint graphs and propagation by path consistency. Alan Mackworth (1977) proposed the AC-3 algorithm for enforcing arc consistency as well as the general idea of combining backtracking with some degree of consistency enforcement. AC-4, a more efficient

arc-consistency algorithm developed by Mohr and Henderson (1986), runs in $O(cd^2)$ worst-case time but can be slower than AC-3 on average cases. The PC-2 algorithm (Mackworth, 1977) achieves path consistency in much the same way that AC-3 achieves arc consistency.

Soon after Mackworth's paper appeared, researchers began experimenting with the trade-off between the cost of consistency enforcement and the benefits in terms of search reduction. Haralick and Elliott (1980) favored the minimal forward-checking algorithm described by McGregor (1979), whereas Gaschnig (1979) suggested full arc-consistency checking after each variable assignment—an algorithm later called MAC by Sabin and Freuder (1994). The latter paper provides somewhat convincing evidence that on harder CSPs, full arc-consistency checking pays off. Freuder (1978, 1982) investigated the notion of k-consistency and its relationship to the complexity of solving CSPs. Dechter and Dechter (1987) introduced directional arc consistency. Apt (1999) describes a generic algorithmic framework within which consistency propagation algorithms can be analyzed, and surveys are given by Bessière (2006) and Barták *et al.* (2010).

Constraint logic programming

Special methods for handling higher-order or global constraints were developed first within the context of **constraint logic programming**. Marriott and Stuckey (1998) provide excellent coverage of research in this area. The *Alldiff* constraint was studied by Regin (1994), Stergiou and Walsh (1999), and van Hoeve (2001). There are more complex inference algorithms for *Alldiff* (see van Hoeve and Katriel, 2006) that propagate more constraints but are more computationally expensive to run. Bounds constraints were incorporated into constraint logic programming by Van Hentenryck *et al.* (1998). A survey of global constraints is provided by van Hoeve and Katriel (2006).

Sudoku has become the most widely known CSP and was described as such by Simonis (2005). Agerbeck and Hansen (2008) describe some of the strategies and show that Sudoku on an $n^2 \times n^2$ board is in the class of *NP*-hard problems.

In 1850, C. F. Gauss described a recursive backtracking algorithm for solving the 8-queens problem, which had been published in the German chess magazine *Schachzeitung* in 1848. Gauss called his method *Tatonniren*, derived from the French word *tâtonner*—to grope around, as if in the dark.

According to Donald Knuth (personal communication), R. J. Walker introduced the term *backtrack* in the 1950s. Walker (1960) described the basic backtracking algorithm and used it to find all solutions to the 13-queens problem. Golomb and Baumert (1965) formulated, with examples, the general class of combinatorial problems to which backtracking can be applied, and introduced what we call the MRV heuristic. Bitner and Reingold (1975) provided an influential survey of backtracking techniques. Brelaz (1979) used the degree heuristic as a tiebreaker after applying the MRV heuristic. The resulting algorithm, despite its simplicity, is still the best method for k-coloring arbitrary graphs. Haralick and Elliott (1980) proposed the least-constraining-value heuristic.

The basic backjumping method is due to John Gaschnig (1977, 1979). Kondrak and van Beek (1997) showed that this algorithm is essentially subsumed by forward checking. Conflict-directed backjumping was devised by Prosser (1993). Dechter (1990a) introduced graph-based backjumping, which bounds the complexity of backjumping-based algorithms as a function of the constraint graph (Dechter and Frost, 2002).

Dependency-directed backtracking

A very general form of intelligent backtracking was developed early on by Stallman and Sussman (1977). Their technique of **dependency-directed backtracking** combines back-

jumping with no-good learning (McAllester, 1990) and led to the development of **truth maintenance systems** (Doyle, 1979), which we discuss in Section 10.6.2. The connection between the two areas is analyzed by de Kleer (1989).

Constraint learning

The work of Stallman and Sussman also introduced the idea of **constraint learning**, in which partial results obtained by search can be saved and reused later in the search. The idea was formalized by Dechter (1990a). **Backmarking** (Gaschnig, 1979) is a particularly simple method in which consistent and inconsistent pairwise assignments are saved and used to avoid rechecking constraints. Backmarking can be combined with conflict-directed backjumping; Kondrak and van Beek (1997) present a hybrid algorithm that provably subsumes either method taken separately.

The method of **dynamic backtracking** (Ginsberg, 1993) retains successful partial assignments from later subsets of variables when backtracking over an earlier choice that does not invalidate the later success. Moskewicz *et al.* (2001) show how these techniques and others are used to create an efficient SAT solver. Empirical studies of several randomized backtracking methods were done by Gomes *et al.* (2000) and Gomes and Selman (2001). Van Beek (2006) surveys backtracking.

Local search in constraint satisfaction problems was popularized by the work of Kirkpatrick *et al.* (1983) on simulated annealing (see Chapter 4), which is widely used for VLSI layout and scheduling problems. Beck *et al.* (2011) give an overview of recent work on jobshop scheduling. The min-conflicts heuristic was first proposed by Gu (1989) and was developed independently by Minton *et al.* (1992). Sosic and Gu (1994) showed how it could be applied to solve the 3,000,000 queens problem in less than a minute. The astounding success of local search using min-conflicts on the n-queens problem led to a reappraisal of the nature and prevalence of "easy" and "hard" problems. Peter Cheeseman *et al.* (1991) explored the difficulty of randomly generated CSPs and discovered that almost all such problems either are trivially easy or have no solutions. Only if the parameters of the problem generator are set in a certain narrow range, within which roughly half of the problems are solvable, do we find "hard" problem instances. We discuss this phenomenon further in Chapter 7.

Konolige (1994) showed that local search is inferior to backtracking search on problems with a certain degree of local structure; this led to work that combined local search and inference, such as that by Pinkas and Dechter (1995). Hoos and Tsang (2006) provide a survey of local search techniques, and textbooks are offered by Hoos and Stützle (2004) and Aarts and Lenstra (2003).

Work relating the structure and complexity of CSPs originates with Freuder (1985) and Mackworth and Freuder (1985), who showed that search on arc-consistent trees works without any backtracking. A similar result, with extensions to acyclic hypergraphs, was developed in the database community (Beeri *et al.*, 1983). Bayardo and Miranker (1994) present an algorithm for tree-structured CSPs that runs in linear time without any preprocessing. Dechter (1990a) describes the cycle-cutset approach.

Since those papers were published, there has been a great deal of progress in developing more general results relating the complexity of solving a CSP to the structure of its constraint graph. The notion of tree width was introduced by the graph theorists Robertson and Seymour (1986). Dechter and Pearl (1987, 1989), building on the work of Freuder, applied a related notion (which they called **induced width** but is identical to tree width) to constraint satisfaction problems and developed the tree decomposition approach sketched in Section 6.5.

Drawing on this work and on results from database theory, Gottlob *et al.* (1999a, 1999b) developed a notion, **hypertree width**, that is based on the characterization of the CSP as a hypergraph. In addition to showing that any CSP with hypertree width w can be solved in time $O(n^{w+1} \log n)$, they also showed that hypertree width subsumes all previously defined measures of "width" in the sense that there are cases where the hypertree width is bounded and the other measures are unbounded.

The RELSAT algorithm of Bayardo and Schrag (1997) combined constraint learning and backjumping and was shown to outperform many other algorithms of the time. This led to AND-OR search algorithms applicable to both CSPs and probabilistic reasoning (Dechter and Mateescu, 2007). Brown *et al.* (1988) introduce the idea of symmetry breaking in CSPs, and Gent *et al.* (2006) give a survey.

The field of **distributed constraint satisfaction** looks at solving CSPs when there is a collection of agents, each of which controls a subset of the constraint variables. There have been annual workshops on this problem since 2000, and good coverage elsewhere (Collin *et al.*, 1999; Pearce *et al.*, 2008).

Comparing CSP algorithms is mostly an empirical science: few theoretical results show that one algorithm dominates another on all problems; instead, we need to run experiments to see which algorithms perform better on typical instances of problems. As Hooker (1995) points out, we need to be careful to distinguish between competitive testing—as occurs in competitions among algorithms based on run time—and scientific testing, whose goal is to identify the properties of an algorithm that determine its efficacy on a class of problems.

The textbooks by Apt (2003), Dechter (2003), Tsang (1993), and Lecoutre (2009), and the collection by Rossi *et al.* (2006), are excellent resources on constraint processing. There are several good survey articles, including those by Dechter and Frost (2002), and Barták *et al.* (2010). Carbonnel and Cooper (2016) survey tractable classes of CSPs. Kondrak and van Beek (1997) give an analytical survey of backtracking search algorithms, and Bacchus and van Run (1995) give a more empirical survey. Constraint programming is covered in the books by Apt (2003) and Fruhwirth and Abdennadher (2003). Papers on constraint satisfaction appear regularly in *Artificial Intelligence* and in the specialist journal *Constraints*; the latest SAT solvers are described in the annual International SAT Competition. The primary conference venue is the International Conference on Principles and Practice of Constraint Programming, often called *CP*.